1

Objekte und Klassen

Wo Begriffe fehlen, stellt ein
Wort zur rechten Zeit sich ein.
Goethe, Faust

Bei der ***objektorientierten Programmierung*** geht es – wie der Name vermuten lässt – um *Objekte*. Leider ist „Objekt“ ein Allerweltswort, das etwa den gleichen Grad von Bestimmtheit hat wie Ding, Sache, haben, tun oder sein. Damit stehen wir vor einem Problem: Ein Wort, das in der Umgangssprache für tausenderlei Dinge stehen kann, muss plötzlich mit einer ganz bestimmten technischen Bedeutung verbunden werden.

Natürlich steht hinter einer solchen Wortwahl auch eine Idee. In diesem Fall geht es um einen *Paradigmenwechsel in der Programmierung*. Während klassischerweise die Algorithmen im Vordergrund standen, also das, was die Programme bei ihrer Ausführung tun, geht es jetzt mehr um Strukturierung der Programme, also um die Organisation der Software. Kurz:

Nicht mehr „*Wie wirds getan?*“ ist die primäre Frage, sondern „*Wer tuts?*“

1.1 Objekte

Um den Paradigmenwechsel von der klassischen zur objektorientierten Programmierung zu erläutern, betrachten wir ein kleines Beispiel. Nehmen wir an, es soll eine Simulation eines Asteroidenfeldes programmiert werden. In der traditionellen Programmierung, der sog. imperativen Programmierung, würde man das in einem Design tun, das in Abbildung 1.1 skizziert ist. Bei diesem Design hat man zwei große, relativ monolithische Programme. Das eine realisiert die astronomischen Berechnungen, das andere zeichnet die Asteroiden auf dem Bildschirm. Beide arbeiten auf einem großen Datenbereich – üblicherweise ein sog. Array –, in dem die Attribute der einzelnen Asteroiden, also Ort, Masse und Geschwindigkeit, gespeichert werden. Dieses Design ist gut geeignet für die Programmierung in einer traditionellen Sprache wie FORTRAN, PASCAL, ADA oder auch C.

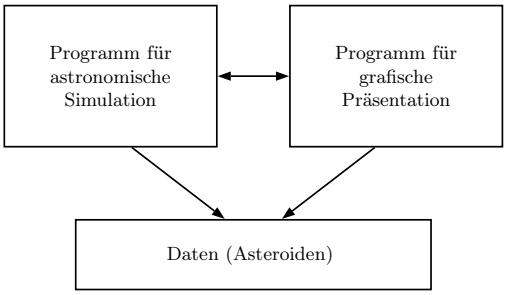
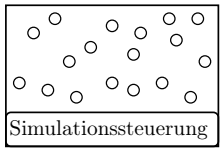


Abb. 1.1. Programmdesign im traditionellen imperativen Stil

In der objektorientierten Programmierung stört man sich primär an den großen monolithischen Programmen. Erfahrungsgemäß sind solche Programme schwer zu warten und nur mühsam an neue Gegebenheiten zu adaptieren. Deshalb löst man sie lieber in kleine überschaubare Einheiten auf.

Für unser obiges Beispiel führt diese Idee auf ein anderes Design. Wir erheben die Asteroiden von schlichten passiven Daten, mit denen etwas gemacht wird, zu aktiven „Objekten“, die selbst etwas tun. Das heißt, jedes Asteroid-Objekt hat nicht nur seine Attribute Ort, Masse und Geschwindigkeit, sondern besitzt auch die Fähigkeit, selbst zu rechnen. Das Programm besteht damit aus einer Ansammlung von Objekten, die sich alle miteinander unterhalten können. Jedes Objekt kann von jedem anderen dessen Masse und Position erfragen, und aus diesen Informationen dann die eigene neue Geschwindigkeit und Position errechnen. Außerdem besitzt jedes dieser Objekte die Fähigkeit, sich auf dem Bildschirm selbst zu zeichnen. Das Ganze wird vervollständigt durch ein Objekt zur Simulationssteuerung, das im Wesentlichen nur dafür sorgt, dass alle Objekte synchron arbeiten.



Dieses Design hat einen unschönen Aspekt. Die beiden Tätigkeiten der astronomischen Simulation und des Zeichnens auf einem Bildschirm haben nichts miteinander zu tun. Deshalb ist es nicht gut, sie in denselben Objek-

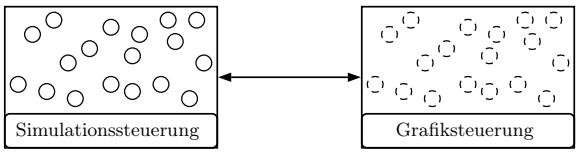


Abb. 1.2. Programmdesign im objektorientierten Stil

ten zu bündeln. Daher ist die beste Lösung eine saubere Aufgabentrennung, wie sie in Abbildung 1.2 skizziert ist. Jetzt gibt es zwei Arten von Objekten,

die eigentlichen Asteroid-Objekte und für jedes von ihnen als „Partner“ ein Grafikobjekt. Die Asteroid-Objekte beherrschen nur noch die Berechnung der astronomischen Gesetze, die zur Simulation gebraucht werden. Die Grafikobjekte können alles, was mit der Darstellung auf dem Bildschirm zusammenhängt. Die notwendigen Daten, vor allem die Position und ggf. auch die Größe erfragen die Grafikobjekte jeweils von ihrem Partner.

Durch diese Trennung von Rechnung und grafischer Darstellung ist das System wesentlich modularer und änderungsfreundlicher geworden. Es sind vor allem diese Eigenschaften, die wesentlich für den Durchbruch des objektorientierten Paradigmas bei der Softwareproduktion verantwortlich sind.

Aus diesem kleinen und noch recht informellen Beispiel können wir schon die zentralen Charakteristika von Objekten ableiten.

Definition (Objekt)

Ein ***Objekt*** wird durch drei Aspekte charakterisiert.

- **Eigenständige Identität**. Ein Objekt kann sich zwar im Lauf der Zeit ändern, das heißt, neue Attributwerte annehmen und ein neues Verhalten zeigen, aber es bleibt immer das gleiche Objekt. Programmiertechnisch wird diese eindeutige und feste Identität durch einen *Namen* (auch *Referenz* genannt) sichergestellt.
- **Zustand**. Zu jedem Zeitpunkt befindet sich das Objekt in einem gewissen „Zustand“. Programmiertechnisch wird das durch sog. *Attribute* realisiert. Das heißt, der Zustand des Objekts ist immer durch die aktuellen Werte seiner Attribute bestimmt.
- **Verhalten**. Ein Objekt ist in der Lage *Aktionen* auszuführen. Das heißt, es kann seinen Zustand (seine Attribute) ändern. Es kann aber auch mit anderen Objekten *interagieren* und sie veranlassen, ihrerseits Aktionen auszuführen. Programmiertechnisch wird das durch sog. *Methoden* realisiert.

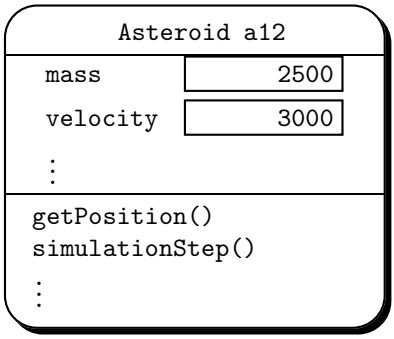
❑

Betrachten wir z. B. ein Auto. Es bleibt dasselbe Fahrzeug, egal ob es gerade steht, fährt, beschleunigt, bremst oder sich überschlägt. Sein Zustand ist durch eine Fülle von Attributen bestimmt; das reicht von kaum veränderlichen Attributen wie Farbe, Gewicht, Motorleistung etc. bis zu sehr flüchtigen Attributen wie Geschwindigkeit, Fahrtrichtung, Motortemperatur usw. Und schließlich gibt es auch eine ganze Reihe von Aktionen, die das Auto seinem Fahrer anbietet, etwa Starten, Beschleunigen, Bremsen, Lenken oder Hupen.

Einige dieser Begriffe sind bei Objekten der realen Welt etwas knifflig. Wenn wir z. B. bei einem Auto die Reifen wechseln oder das Radio austauschen, werden wir sicher sagen, dass es immer noch das gleiche Auto ist – von dem wir allerdings einen Teil ausgetauscht haben. Wenn wir aber einen Totalschaden hatten und nur das Radio in das nächste Auto retten, werden wir wohl kaum davon reden, dass wir immer noch unser altes Auto haben – nur mit gewissen ausgetauschten Teilen. Bei programmiertechnischen Objek-

ten gibt es solche diffusen Situationen aber nicht: Hier ist die Identität von Objekten immer klar geregelt.

Grafisch stellen wir Objekte häufig folgendermaßen dar:



Diese Darstellung entspricht den drei Teilen des Objektbegriffs.

- Oben steht der *Name* des Objekts (`a12`) und um welche Art von Objekt es sich handelt (`Asteroid`).
- Den nächsten Block bilden die *Attribute* des Objekts. Dabei geben wir jeweils die Attributbezeichnung (z. B. `velocity`) an und tragen den aktuellen Wert des Attributs in den zugehörigen „Slot“ ein (z. B. `3000 km/h`).
- Den letzten Block bilden die *Methoden* des Objekts, in unserem Beispiel `getPosition` und `simulationStep`. Die Klammern deuten dabei an, dass es sich um Methoden handelt.

1.2 Beschreibung von Objekten: Klassen

In unserer Simulation haben wir Hunderte, wenn nicht Tausende von Asteroiden. Sie alle einzeln zu programmieren wäre offensichtlich ein hoffnungsloses Unterfangen. Und es wäre auch ziemlich dumm. Denn die Programme wären alle identisch. Wir brauchen also einen Trick, mit dem wir nur einmal aufschreiben müssen, wie unsere Asteroid-Objekte aussehen sollen, und mit dem wir dann beliebig viele Objekte schaffen können.

Dieser Trick ist jedem Ingenieur bekannt. Man nennt ihn *Bauplan* oder *Blaupause*. Wenn man einen Plan für einen Zylinderkopf hat, lassen sich nach dieser Anleitung beliebig viele Zylinderköpfe produzieren. Aber auch in der Einzelfertigung hat sich das bewährt: Selbst wenn man nur ein einzelnes Haus bauen will, sollte man sich vorher vom Architekten einen Plan zeichnen lassen.

Diese fundamentale Rolle von Bauplänen hat die Informatik von den Ingenieuren und Architekten übernommen. Wenn wir Objekte haben wollen, sollten wir sie nicht ad hoc basteln, sondern systematisch planen. Und wenn wir dann einen Plan haben, können wir damit beliebig viele Objekte automatisch herstellen – oder auch nur ein einziges, je nachdem, was wir brauchen. Solche Baupläne heißen in der objektorientierten Programmierung *Klassen*.

Definition (Klasse)

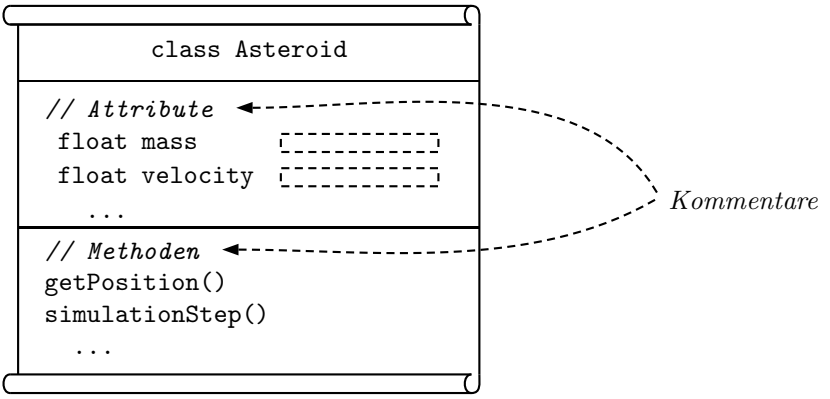
Eine ***Klasse*** ist ein „Bauplan“ für gleichartige Objekte. Sie beschreibt

– welche **Attribute** die Objekte haben;

– welche **Methoden** die Objekte haben.

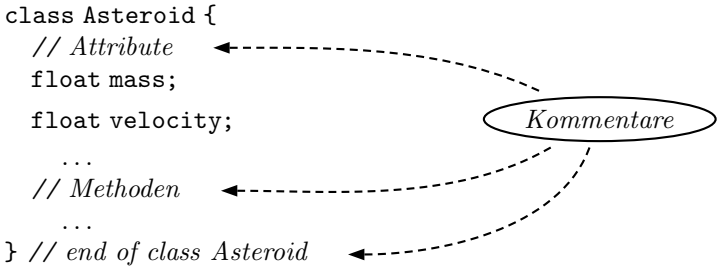
❑

Um die Tatsache zu unterstreichen, dass Klassen als Blaupausen für Objekte dienen, wählen wir eine entsprechende grafische Darstellung.



- Oben steht der *Name* der Klasse.
- Danach kommen die Namen der *Attribute*. Diese versehen wir auch noch mit dem Typ ihrer Werte. In unserem Beispiel sind die Attribute `mass` und `velocity` jeweils sog. Floating-Point-Zahlen.
- Den letzten Block bilden die *Methoden*. Dabei ist das Bild allerdings nur eine grobe Skizze. Im tatsächlichen JAVA-Programm steht an dieser Stelle nicht nur der Name der Methode, sondern der gesamte Programmtext.

In JAVA-Notation sieht das so aus:



Dieses Minibeispiel zeigt die Grundstruktur der Klassennotation in JAVA. Sie wird eingeleitet mit dem Schlüsselwort `class`, gefolgt vom Namen der Klasse. Die eigentliche Definition erfolgt dann im ***Klassenrumpf***, der in die Klammern `{ ... }` eingeschlossen ist.

Klasse
`class «Name» { «Klassenrumpf» }`

In dem Beispiel sieht man auch einige andere Dinge, auf die wir später noch genauer eingehen werden.

- *Kommentare* werden mit einem doppelten Schrägstrich `//` eingeleitet. Alles was zwischen diesem Symbol und dem Ende der Zeile steht, wird vom Compiler ignoriert und kann deshalb zur Erläuterung und Dokumentation für den menschlichen Leser benutzt werden.
  Da in JAVA schrecklich viel mit dem Klammerpaar `{ ... }` erledigt wird, ist es eine nützliche Konvention, bei der schließenden Klammer als Kommentar anzugeben, *was* geschlossen wird.
- *Attribute* schreibt man in der Form «`Art`» «`Name`», also z. B. `float mass`. Die Art (auch *Typ* genannt) `float` ist in JAVA vordefiniert.

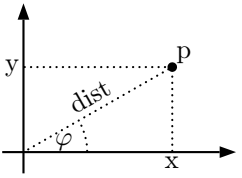
Da das Asteroidenbeispiel recht groß geraten würde, wollen wir uns im Folgenden lieber mit etwas einfacheren und kürzeren Beispielen beschäftigen.

1.3 Klassen und Konstruktormethoden

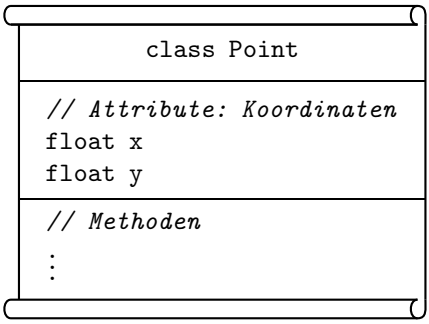
Nach den bisherigen allgemeinen Vorüberlegungen zu Objekten und Klassen wollen wir uns jetzt mit ihrer konkreten Programmierung in der Sprache JAVA befassen. Um das Ganze greifbarer zu machen, tun wir dies im Rahmen eines einfachen Beispiels.

1.3.1 Beispiel: Punkte im $\mathbb{R}^2$

Zur Einführung der JAVA-Konzepte verwenden wir ein Beispiel, das wir ziemlich vollständig ausarbeiten und in JAVA aufschreiben können. Nehmen wir an, wir wollen Programme schreiben, mit denen wir ein bisschen Geometrie im $\mathbb{R}^2$ treiben können. Dazu brauchen wir auf jeden Fall erst einmal Punkte. Ein Punkt ist durch seine x- und y-Koordinaten charakterisiert. Außerdem wollen wir ein paar Methoden zur Verfügung haben, z. B. um den Winkel und die Distanz vom Nullpunkt zu berechnen. Im Folgenden werden wir diese Klasse (und ein paar andere) Stück für Stück einführen und dabei einen ersten Einblick in die Sprachkonzepte von JAVA erhalten.



Der folgende Bauplan zeigt, dass die Objekte der Klasse `Point` zwei Attribute besitzen. Sie haben die Namen `x` und `y` und sind vom Typ `float`. Es gibt auch eine Reihe von Methoden, die wir aber erst später einführen werden.



1.3.2 Klassen in JAVA

Wir wollen uns jetzt aber nicht mit abstrakten Bildern von Bauplänen begnügen, sondern auch die konkrete Programmierung in JAVA ansehen.

```
class Point {
  // Attribute: Koordinaten
  float x;
  float y;
  // Methoden
  ⋮
} // Point
```

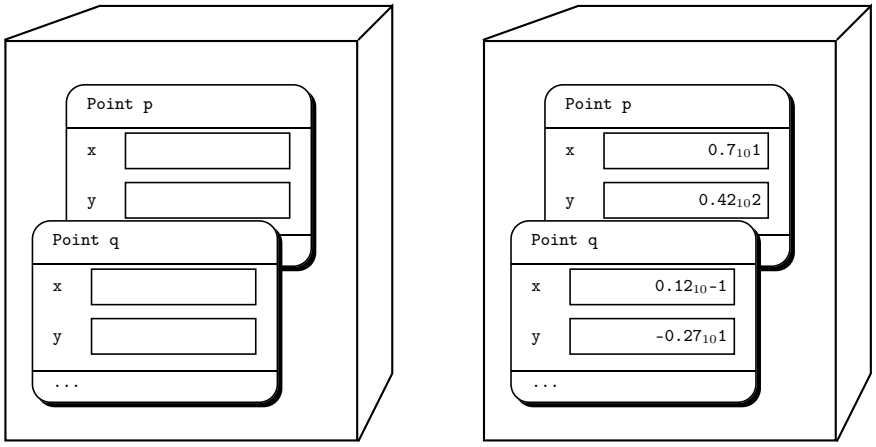
Die nächste Frage ist: Wenn wir den Bauplan haben, wie kommen wir zu den konkreten Objekten? Dafür stellt JAVA einen speziellen Operator zur Verfügung: `new`. Wir können also schreiben

```
Point p = new Point();
Point q = new Point();
```

Damit entstehen zwei Objekte mit den Namen `p` und `q`. (Das ist zumindest eine hinreichend akkurate Intuition für den Augenblick. Genauer werden wir das in einem späteren Kapitel noch studieren.) Wir können uns das so vorstellen, dass mit den beiden `new`-Anweisungen im Computer zwei konkrete Objekte entstanden sind. Diese Situation ist auf der linken Seite von Abbildung 1.3 skizziert.

Aber diese Objekte sind noch unbrauchbar, denn ihre Slots für die Attribute sind noch leer. Das heißt, wir haben zwar zwei Objekte im Rechner kreiert, aber diese Objekte sind noch nicht das, was wir uns unter Punkten vorstellen. Damit sie ihren Zweck erfüllen können, müssen wir sie mit Koordinatenwerten versehen. Das geschieht – nach dem `new` – in folgender Form:

```
Point p = new Point();   // kreiere Punkt p
p.x = 7f;                // setze x-Koordinate von p
p.y = 42f;               // setze y-Koordinate von p
Point q = new Point();   // kreiere Punkt q
q.x = 0.012f;            // setze x-Koordinate von q
q.y = -2.7f;             // setze y-Koordinate von q
```



(a) Nach dem `new` (b) Nach den Attributsetzungen

Abb. 1.3. Effekt von `new` und Attributsetzung im Rechner

Diese sog. *Punktnotation* findet sich überall in JAVA. Die Namen der Attribute (und auch die der Methoden) dienen als *Selektoren*. Wenn in der Klasse `Point` ein Attribut mit dem Namen `x` eingeführt wurde, und wenn `p` ein Objekt der Art `Point` ist, dann wird mit der Selektion `p.x` der entsprechende Slot von `p` bezeichnet. Die Anweisung `p.x = 7f` trägt damit den Wert `7` in den zugehörigen Slot von `p` ein. Der Effekt der zwei `new`-Anweisungen und der vier Attributsetzungen ist auf der rechten Seite von Abbildung 1.3 illustriert.

Übrigens: Wie man hier auch noch sieht, muss man hinter konkrete Zahlen der Art `float` in JAVA ein '`f`' setzen, also z. B. '`7f`' (s. Abschnitt 2.1).

Außerdem kann man auch sehen, dass in JAVA jede Anweisung mit einem Semikolon '`;`' abgeschlossen wird.

Anmerkung: In dem Bild Abbildung 1.3(b) haben wir eine spezielle Eigenschaft von Computern berücksichtigt. In der Maschine werden sog. Gleitpunktzahlen (engl.: Floating point numbers) in normalisierter Form dargestellt. Das heißt, sie werden z. B. als $0.27_{10}1$ *oder* $0.12_{10}-1$ *gespeichert, also immer in der Form* $0.x\ldots x_{10}e\ldots e$*, wobei die sog. Mantisse* $x\ldots x$ *keine führende Nullen hat und die tatsächliche Position des Dezimalpunkts im sog. Exponenten* $e\ldots e$ *festgehalten wird.*

1.3.3 Konstruktor-Methoden

Unser Beispiel zeigt ein wichtiges Phänomen der Programmierung mit Objekten. Mittels `new` werden „blanke" Objekte kreiert, also Objekte ohne Attributwerte. Solche Objekte sind fast immer nutzlos. Deshalb dürfen wir nie vergessen, sofort nach dem Kreieren der Objekte ihre Attribute zu setzen.

Damit haben wir aber eine potenzielle Fehlersituation geschaffen. Menschen sind vergesslich, und Programmierer sind auch nur Menschen. Also wird

es immer wieder vorkommen, dass jemand das Setzen der Attribute vergisst. Die resultierenden Fehlersituationen können subtil und schwer zu finden sein.

Die Lösung dieses Problems ist offensichtlich. Man muss dafür sorgen, dass die Erzeugung des Objekts und die Setzung seiner Attribute gleichzeitig passieren. Wir würden also gerne schreiben

```
Point p = new Point(7f, 42f);
Point q = new Point(0.012f, -2.7f);
```

Zu diesem Zweck stellt JAVA die sog. ***Konstruktormethoden*** zur Verfügung. Man schreibt sie wie im folgenden Beispiel illustriert.

```
class Point {
  // Attribute: Koordinaten
  float x;
  float y;
  // Konstruktor-Methode
  Point ( float x, float y ) {
    this.x = x;                 // setze Attribut x
    this.y = y;                 // setze Attribut y
  } // Point
  // Methoden
  ...
} // class Point
```

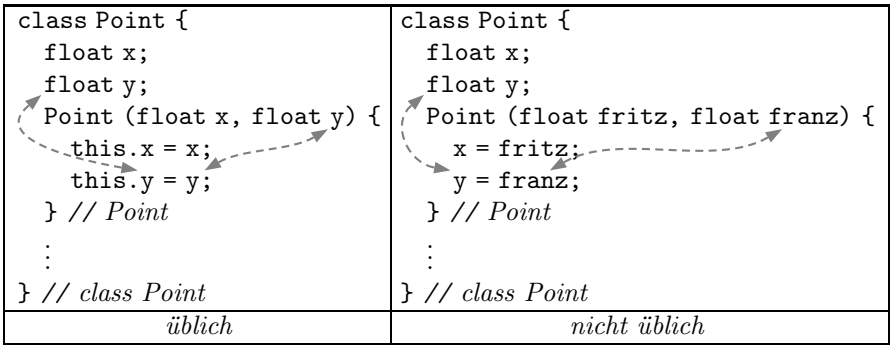
Das bedarf einiger Erklärung. Zunächst sieht man, dass die Konstruktormethode genauso heißt wie die Klasse selbst, in unserem Beispiel also `Point`. Die sog. *Parameter* – in unserem Fall haben wir sie `x` und `y` genannt – werden bei der Anwendung durch die jeweiligen Werte ersetzt. Das heißt

```
new Point(7f, 42f)     entspricht     this.x = 7f;
                                      this.y = 42f;
```

Damit bleibt nur noch zu klären, was es mit diesem ominösen `this` auf sich hat. Erinnern wir uns: Wir müssen die Attributwerte in die Slots der jeweiligen Objekte eintragen. Wenn wir Objekte wie `p` und `q` haben, dann beziehen wir uns auf diese Slots mit der Selektorschreibweise `p.x`, `q.x` etc. Aber die Klasse dient ja als Bauplan für *alle* Objekte; deshalb brauchen wir innerhalb der Programmierung der Klasse selbst ein anderes Mittel, um uns auf die Attributslots zu beziehen. Und das ist eben `this`. Damit gilt

```
Point p = new Point(7f, 42f)   entspricht   Point p = new Point();
                                            p.x = 7f;
                                            p.y = 42f;
```

Programmierer sind faule Menschen. Deshalb streben sie nach Abkürzungen. Und deshalb wären sie gerne den Zwang los, immer `this` schreiben zu müssen. JAVA kommt dieser Faulheit entgegen. Wir können die Konstruktormethode nämlich auch anders schreiben.



Auf der linken Seite heißen die Parameter genauso wie die Attribute; deshalb muss man z. B. mit `this.y` klarmachen, dass das Attribut gemeint ist. Der Name `y` alleine bezieht sich nämlich auf den – näher stehenden – Parameter.

Auf der rechten Seite heißen die Parameter anders als die Attribute. Deshalb gibt es z. B. in `y = franz` für das `y` gar keinen anderen Kandidaten als das Attribut. Allerdings wäre auch `this.y = franz` erlaubt gewesen.

Im Übrigen zeigt die Wahl der etwas flapsigen Namen `fritz` und `franz`, dass man Parameter beliebig nennen darf. Dem Aufruf `new Point(7f, 42f)` sieht man diese Namen ohnehin nicht mehr an. Man kann das ausnutzen, um die Parameternamen möglichst einprägsam und selbsterklärend zu wählen. (`fritz` *und* `franz` *sind daher eine miserable Wahl!*)

In der JAVA-*Community* hat sich die Konvention eingebürgert, bei den Konstruktormethoden die Parameter genauso zu nennen wie die Attribute, die mit ihnen gesetzt werden sollen. Deshalb entspricht die linke Variante mit `this` den üblichen Gewohnheiten.

Definition (Konstruktor-Methode)

Eine ***Konstruktormethode*** heißt genauso wie die Klasse selbst. Sie wird üblicherweise dazu verwendet, bei der Generierung von Objekten mittels `new` auch gleich die Attribute geeignet zu setzen.

Als Konvention hat sich eingebürgert, die Parameter der Methode so zu nennen wie die entsprechenden Attribute. Deshalb wird das Schlüsselwort `this` benötigt, um Attribute und Parameter unterscheiden zu können. ❑

Jetzt wird klar, weshalb wir ganz am Anfang, als wir noch keine Konstruktormethode in der Klasse `Point` eingeführt hatten, schreiben mussten

```
Point p = new Point();
```

Das `Point` hinter `new` war gar nicht der Klassenname! Es war von Anfang an eine Konstruktormethode – allerdings eine ganz spezielle. Denn JAVA kreiert *automatisch* zu jeder Klasse eine Konstruktormethode, vorausgesetzt der Programmierer schreibt nicht selbst eine. Diese automatisch erzeugte Konstruktormethode hat *keine* Parameter, was sich in dem leeren Klammerpaar bei `new Point()` zeigt.

Diese automatisch erzeugte Methode gibt es aber nicht mehr, sobald man selbst eine Konstruktormethode in der Klasse programmiert. In unserer jetzigen Form der Klasse `Point` wäre die Anweisung `Point p = new Point()` also falsch! Der Compiler würde sich beschweren, dass er eine Methode `Point()` – also ohne Parameter – nicht kennt.

Was ist, wenn man so eine „nackte" Methode aber trotzdem braucht? Kein Problem – JAVA erlaubt auch die Definition mehrerer Konstruktormethoden in einer Klasse. Die einzige Bedingung ist, dass sie alle verschiedenartige Parameter haben müssen. Man spricht dann von ***Überlagerung*** (engl.: ***Overloading***) von Methoden (s. Abschnitt 3.1.4).

```
class Point {
  // Attribute: Koordinaten
  float x;
  float y;
  // Konstruktor-Methoden
  Point () {}                    // ohne Parameter
  Point ( float x ) {            // gleiche Koordinaten
    this.x = x;  this.y = x;
  } // Point
  Point ( float x, float y ) {   // verschiedene Koordinaten
    this.x = x;   this.y = y;
  } // Point
  ...
} // class Point
```

Die erste dieser drei Konstruktormethoden hat einen leeren Rumpf – sie tut gar nichts! (Das ist erlaubt.) Die zweite besetzt beide Koordinaten gleich. Damit ist also `new Point(1f)` gleichwertig zu `new Point(1f,1f)`.

Programm 1.1 Die Klasse `Point` (Teil 1)

```
class Point {
  // Attribute: Koordinaten
  float x;
  float y;
  // Konstruktor-Methode
  Point ( float x, float y ) {
    this.x = x;                  // setze Attribut x
    this.y = y;                  // setze Attribut y
  } // Point
  // Methoden
  ...
} // class Point
```

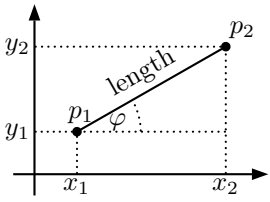
Aber für das Weitere wollen wir uns auf den üblichen Fall konzentrieren, dass es eine Konstruktormethode `Point` gibt, und dass diese die beiden Koordinaten setzt. Das Programmfragment 1.1 fasst unseren bisherigen Entwicklungsstand bei der Klasse `Point` zusammen, von dem wir im Folgenden ausgehen werden.

1.4 Objekte als Attribute von Objekten

Im Beispiel `Point` hatten wir als Attribute nur Werte der Art `float`, also elementare Werte, die von JAVA vorgegeben sind und in Computern unmittelbar gespeichert werden können. Das muss aber nicht so sein.

1.4.1 Beispiel: Linien im $\mathbb{R}^2$

Nur mit Punkten zu arbeiten wäre etwas langweilig. Als Mindestes sollte man noch Linien zur Verfügung haben. Wie in der Geometrie üblich, stellen wir Linien durch ihre beiden Endpunkte dar. Damit haben wir gegenüber unserem Beispiel `Point` eine neue Situation: Jetzt haben die Attribute nicht mehr eine von JAVA vorgegebene Art wie `float`, sondern eine von uns selbst definierte Klasse, nämlich `Point`. Auf die weiteren Aspekte der Klasse, z. B. die Methoden für Steigungswinkel und Länge, gehen wir erst später ein. Grafisch stellen wir die Klasse mit folgendem „Bauplan“ dar.




```
┌──────────────────────────────────────┐
│            class Line               │
├──────────────────────────────────────┤
│ // Attribute: Endpunkte              │
│ Point p1                             │
│ Point p2                             │
├──────────────────────────────────────┤
│ // Konstruktormethode                │
│ Line ( Point p1, Point p2 )          │
│ // andere Methoden                   │
│ ⋮                                    │
└──────────────────────────────────────┘
```

Die Aufschreibung in JAVA-Notation sollte jetzt keine Probleme machen.[1]

[1] Die Arbeitsweise dieses Programms wird in einigen Folien illustriert, die man von der begeleitenden Web-Seite des Buches herunterladen kann. (Details findet man in Abschnitt A.10 im Anhang.)

```
class Line {
  // Attribute: Endpunkte
  Point p1;
  Point p2;
  // Konstruktormethode
  Line ( Point p1, Point p2 ) {
    this.p1 = p1;                    // setze Attribut p1
    this.p2 = p2;                    // setze Attribut p2
  } // Line
  // andere Methoden
    ⋮
}// class Line
```

Wenn wir ein Objekt der Art `Line` kreieren wollen, sieht das z. B. so aus;

```
Point p = new Point(1f,1f);
Point q = new Point(2f,3f);
Line l = new Line(p,q);
```

Was geschieht hier im Computerspeicher? In Abbildung 1.4 ist das illustriert. Wir haben zunächst zwei Objekte der Art `Point` erzeugt. Diese befinden sich

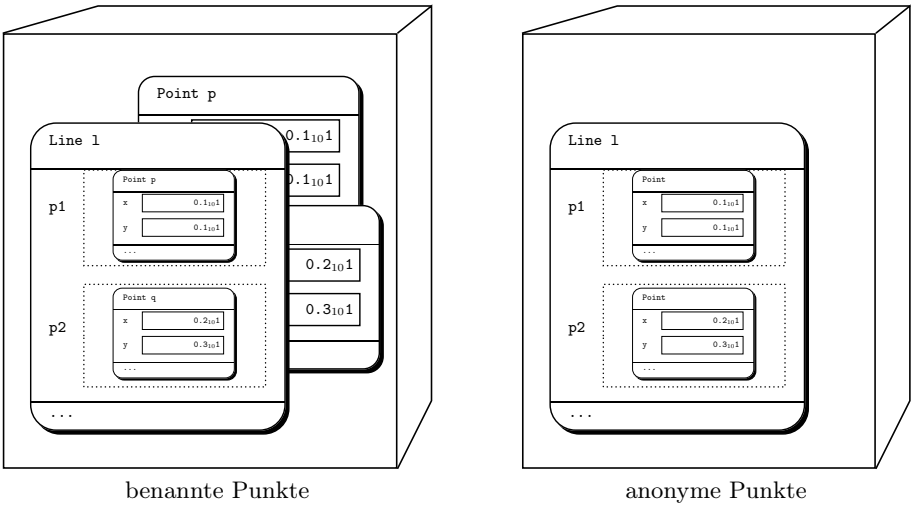


Abb. 1.4. Effekt im Computer

im Speicher unter den Namen `p` und `q`. Dann erzeugen wir ein weiteres Objekt der Art `Line` und speichern es unter dem Namen `l`. Die Attribute dieses Objekts `l` sind jetzt aber keine elementaren Werte, sondern die zuvor erzeugten Objekte `p` und `q`. Das heißt, Objekte können als Attribute wieder Objekte haben.

1.4.2 Anonyme Objekte

Wir brauchen die beiden Punkte nicht unbedingt vorher einzuführen und zu benennen. Als Variante können wir sie auch direkt bei der Kreierung der Linie `l` mit erzeugen:

```
Line l = new Line ( new Point(1f,1f), new Point(2f,3f) );
```

Hier werden zwei anonyme Objekte der Art `Point` erzeugt und sofort als Attribute in das ebenfalls neu erzeugte Objekte `l` der Art `Line` eingetragen. Was bedeutet das? Wir können die beiden Punkte im Programm nicht mehr direkt ansprechen, sondern nur noch über das Objekt `l`. Wir müssen also schreiben `l.p1` oder `l.p2`, um an die Punkte heranzukommen. Die Attribute der Punkte werden dann über mehrfache Selektion wie z. B. `l.p1.x` oder `l.p2.y` erreicht.

Auch hier halten wir im Programmfragment 1.2 wieder den Entwicklungsstand der Klasse `Line` fest, von dem wir im Weiteren ausgehen werden.

Programm 1.2 Die Klasse `Line` (Teil 1)

```
class Line {
  // Attribute: Endpunkte
  Point p1;
  Point p2;
  // Konstruktormethode
  Line ( Point p1, Point p2 ) {
    this.p1 = p1;
    this.p2 = p2;
  } // Line
  // andere Methoden
    ⋮
}// class Line
```

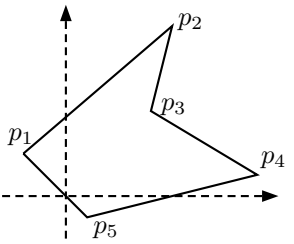
1.5 Objekte in Reih und Glied: Arrays

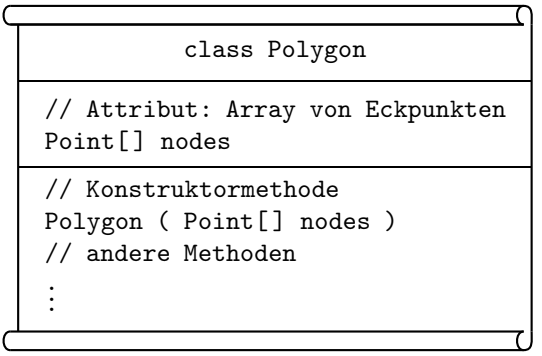
Eine Linie hat zwei Punkte. Ein Dreieck hat drei, ein Viereck vier, ein Fünfeck fünf und so weiter. Man kann sich gut vorstellen, wie die Klasse `Line` sich entsprechend zu Klassen `Triangle`, `Quadrangle`, `Pentagon` etc. verallgemeinern lässt, die jeweils die entsprechende Anzahl von Attributen der Art `Point` haben. Aber was machen wir, wenn wir allgemeine *Polygone* beschreiben wollen, die beliebig viele Punkte haben können?

Dazu gibt es in JAVA– wie in den meisten anderen Programmiersprachen – ein vorgefertigtes Konstruktionsmittel: die sog. *Arrays*. Unserer bisherigen Übung folgend wollen wir auch diese wieder am konkreten Beispiel einführen.

1.5.1 Beispiel: Polygone im $\mathbb{R}^2$

Ein Polygon ist ein Linienzug. Es läge daher nahe, Polygone als Folgen von Linien zu beschreiben; dann hat man aber die Randbedingung, dass der Endpunkt der einen Linie immer mit dem Anfangspunkt der nächsten Linie übereinstimmen muss. Einfacher ist es deshalb, die ansonsten gleichwertige Darstellung als Folge der Eckpunkte zu wählen. Außerdem betrachten wir nur *geschlossene* Polygone, bei denen die Anfangs- und Endpunkte jeweils übereinstimmen. Damit kann z. B. ein Fünfeck als Polygon mit fünf Eckpunkten beschrieben werden. Das können wir wieder in der Form unserer „Baupläne“ darstellen.





Die Aufschreibung in JAVA-Notation ist im Prinzip genauso, wie wir es schon bei `Point` und `Line` kennen gelernt haben. Das einzig Neue sind die leeren eckigen Klammern bei `Point[]`, die offensichtlich der Trick sind, mit dem wir die Idee „eine Folge von vielen Elementen“ erfassen. Man spricht dann von einem *Array*. Programm 1.3 enthält die entsprechenden Definitionen.

Programm 1.3 Die Klasse `Polygon` (Teil 1)

```
class Polygon {
  // Attribute: Array von Eckpunkten
  Point[] nodes;
  // Konstruktormethode
  Polygon ( Point[] nodes ) {
    this.nodes = nodes;            // setze Attribut nodes
  } // Polygon
  // andere Methoden
    ⋮
}// class Polygon
```

Anmerkung: Vorsorglich sollte hier angemerkt werden, dass die Attributsetzung `this.nodes=nodes` *in der Konstruktormethode vom Prinzip her schon in Ordnung ist. Allerdings werden wir in einem späteren Kapitel (nämlich Kapitel 16) sehen, dass es subtile Unterschiede zu Attributen der Art* `float` *gibt. Aber für den Anfang können wir diese Unterschiede ignorieren.*

Wie kann man ein Polygon erzeugen? Zunächst braucht man genügend viele Punkte. Dann muss daraus ein Array gemacht werden, den man der Konstruktormethode des Polygons übergibt. Das sieht in JAVA z. B. folgendermaßen aus.

```
Point p1 = new Point(-2f, 2f);
Point p2 = new Point(5f, 8f);
Point p3 = new Point(4f, 4f);
Point p4 = new Point(9f, 1f);
Point p5 = new Point(1f, -1f);
Point[] points = { p1, p2, p3, p4, p5 };
Polygon poly = new Polygon( points );
```

Diese Schreibweise zeigt, dass man einen Array von Elementen in der Notation {x_1, ..., x_n} schreiben kann. Übrigens ist es hier genauso wie bei den Eckpunkten einer Linie; man muss die Punkte nicht unbedingt explizit benennen, sondern kann sie auch anonym lassen. Das sieht dann so aus:

```
Polygon poly = new Polygon( new Point[] { new Point(-2f, 2f),
                                          new Point(5f, 8f),
                                          new Point(4f, 4f),
                                          new Point(9f, 1f),
                                          new Point(1f, -1f)  } )
```

Man beachte, dass man die Angabe `new Point[]` vor den eigentlichen Elementen `{...}` nicht weglassen darf (weil JAVA sonst nicht weiß, dass die Klammern einen Array bedeuten).

Aus unseren fünf Punkten lassen sich auch andere Polygone basteln. Zum Beispiel:

```
Polygon poly1 = new Polygon( new Point[] { p1, p3, p2, p4, p5 } );
Polygon poly2 = new Polygon( new Point[] { p1, p2, p4, p5, p3 } );
Polygon poly3 = new Polygon( new Point[] { p1, p2, p4, p5 } );
Polygon poly4 = new Polygon( new Point[] { p1, p2, p4 } );
```

Im Folgenden wollen wir uns etwas genauer mit dem Sprachmittel der Arrays befassen – jedenfalls in einer ersten Ausbaustufe.

1.5.2 Arrays: Eine erste Einführung

Häufig müssen wir eine Ansammlung von Werten betrachten, also z.B. eine Messreihe, eine Kundenliste oder eine Folge von Worten. Das lässt sich in Programmiersprachen auf vielfältige Weise beschreiben. Die einfachste Form ist der sog. „Array“.

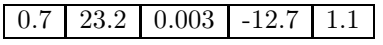
Definition (Array)

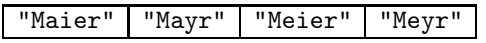
Arrays sind (in JAVA) durch folgende Eigenschaften charakterisiert:

- Ein Array ist eine *geordnete Kollektion von Elementen.*
- Alle Elemente müssen den gleichen Typ haben, der als ***Basistyp*** des Arrays bezeichnet wird.
- Die Anzahl n der Elemente im Array wird als seine ***Länge*** bezeichnet.
- Die Elemente im Array sind von $0, \ldots, n-1$ durchnummeriert.

❑

Bildlich können wir uns z.B. einen Array von Zahlen oder einen Array von Strings folgendermaßen vorstellen:



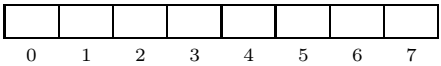


Array-Deklaration. Die Notation orientiert sich an dem, was sich in Programmiersprachen für Arrays allgemein etabliert hat. Mit „`float[]`“ (lies: float-Array) bezeichnet man z.B. den Typ der Arrays über dem Basistyp `float`, mit „`String[]`“ (lies: String-Array) den Typ der Arrays über dem Basistyp `String` und mit „`Point[]`“ (lies: Point-Array) den Typ der Arrays über dem Basistyp `Point`. Die folgenden Beispiele illustrieren diese Notation:

1. Ein `float`-Array `a` mit Platz für 8 Zahlen wird durch folgende Deklaration eingeführt:

   ```
   float[] a = new float[8];
   ```

   Im Ergebnis hat man einen „leeren“ Array mit 8 Plätzen:

2. Ein Array `b` mit Platz für 100 Strings wird so deklariert:

   ```
   String[] b = new String[100];
   ```

3. Manchmal will man einen Array sofort mit konkreten Werten besetzen (also nicht nur Platz vorsehen). Dafür gibt es eine bequeme Abkürzungsnotation: Einen Array mit den ersten fünf Primzahlen kann man folgendermaßen deklarieren (wobei `int` für den Typ der ganzen Zahlen steht):

   ```
   int[] primzahlen = { 2, 3, 5, 7, 11 };
   ```

   Einen Array mit vier Texten erhält man z. B. so:

   ```
   String[] kartenFarben = { "kreuz", "pik", "herz", "karo" };
   ```

Mit dieser Notation werden die Länge und der Inhalt des Arrays gleichzeitig festgelegt.

Array-Selektion. Um einzelne Elemente aus einem Array zu selektieren, verwendet man die Klammern `[...]`. *Man beachte, dass die Indizierung bei 0 anfängt!* Für die obigen Beispiele können wir z.B. folgende Selektionen benutzen:

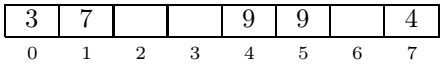
```
primzahlen[0]        // liefert '2'
primzahlen[1]        // liefert '3'
primzahlen[4]        // liefert '11'
kartenFarben[0]      // liefert "kreuz"
```

Wenn man versucht, auf ein Element außerhalb des Indexbereichs des Arrays zuzugreifen – also z. B. `primzahlen[5]` oder `kartenFarben[-1]` – führt das auf einen **Fehleralarm**. (Dieser Alarm hat in JAVA den schönen Namen `ArrayIndexOutOfBoundsException`).[2]

Setzen von Array-Elementen. Die obige Form der kompakten Setzung von Array-Elementen, wie bei den Beispielen `primzahlen` und `kartenFarben`, ist nicht immer möglich oder adäquat. Deshalb kann man Array-Elemente auch einzeln setzen.

```
int[] a = new int[8];    // leerer Array
a[0] = 3;                // erstes Element setzen
a[1] = 7;                // zweites Element setzen
a[4] = 9;                // fünftes Element setzen
a[5] = 9;                // sechstes Element setzen
a[7] = 4;                // achtes Element setzen
```

Als Ergebnis hat man einen Array der Länge 8, in dem fünf Elemente besetzt und die anderen drei leer sind:



Länge des Arrays. Die Länge eines Arrays kann man über das Attribut `length` erfahren:

```
kartenFarben.length    // liefert den Wert 4
a.length               // liefert den Wert 8
```

Man beachte aber, dass der maximale Index um eins kleiner ist als die Länge, also z. B. höchstens `kartenFarben[3]` erlaubt ist – eine beliebte Quelle steter Programmierfehler!

[2] Auf die generelle Behandlung von *Eceptions* gehen wir erst in einem späteren Kapitel ein.

1.6 Zusammenfassung: Objekte und Klassen

Das zentrale Programmiermittel von JAVA sind *Klassen*. Sie werden in folgender Form geschrieben:

```
class «Name» {
  «Attribute»
  «Konstruktormethoden»
  «weitere Methoden»
}
```

Dabei dürfen die verschiedenen Bestandteile in beliebiger Reihenfolge stehen, aber die obige Gruppierung hat sich bewährt und wird deshalb von uns – und auch den meisten JAVA-Programmierern – grundsätzlich so eingehalten.

Klassen fungieren als „Baupläne“ für *Objekte*. Die einzelnen Objekte werden dabei mit Hilfe des `new`-Operators erzeugt.

```
new «Konstruktor» ( «Argumente» )
```

Häufig wird dem Objekt bei dieser Gelegenheit auch gleich ein expliziter Name gegeben:

```
«KlassenName» «objektName» = new «Konstruktor» ( «Argumente» );
```

Man beachte die – unter JAVA-Programmierern übliche – Konvention, dass Klassennamen groß- und Objektnamen kleingeschrieben werden.

Zu jeder Klasse gehört mindestens eine *Konstruktormethode*. Sie heißt genauso wie die Klasse. Üblicherweise werden in dieser Konstruktormethode die Anfangswerte der Attribute für das zu kreierende Objekt mitgegeben. Wenn man keine solche Methode programmiert, dann generiert JAVA automatisch einen parameterlosen Konstruktor.

Wenn man eine Kollektion von vielen Elementen braucht, dann sind ein erstes und einfaches Sprachmittel dafür die *Arrays*. Arrays werden durch eckige Klammern notiert, also z. B. `float[] a` oder `Point[] a`. Erzeugt werden sie entweder uninitialisiert in einer Form wie `new float[«Länge»]` oder initialisiert in der Form $\texttt{\{}x_1\texttt{, ..., }x_n\texttt{\}}$. Der Zugriff erfolgt in der Form `a[i]`, die Zuweisung entsprechend `a[i]=...`. Die Länge eines Arrays erhält man in der Form `a.length`.

Anmerkung: Im Software Engineering gibt es inzwischen eine weit verbreitete Notation zur grafischen Darstellung von Klassen, Objekten und ihren Beziehungen, nämlich UML *[54, 63]. Wir verzichten hier aber darauf, neben* JAVA *gleich noch eine zweite Notation einzuführen, und beschränken uns auf intuitive Bilder, die die „Blaupausen-Metapher“ widerspiegeln.*